Operandi in memoria

- La memoria principale è usata per dati composti
  - Array, strutture, dati dinamici
- Per effettuare operazioni aritmetiche
  - Caricare i valori dalla memoria nei registri
  - Copiare il risultato dal registro alla memoria
- La memoria è indirizzata al byte
  - Ogni indirizzo identifica un byte su 8 bit
- Le parole sono allineate in memoria
  - Un indirizzo deve essere un multiplo di 4
- MIPS è *Big Endian*
  - Il byte più significativo è nell'indirizzo più basso di una parola
  - *Little Endian*: il byte meno significativo è nell'indirizzo più basso di una parola

Esempio 1

- Codice C:

```
g = h + A[8]
```

  - `g` in `$s1`, `h` in `$s2`, indirizzo base di `A` in `$s3`
- Codice MIPS compilato:
  - L'indice 8 richiede un offset di 32 byte
    - 4 byte per parola

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset — registro base

Esempio 2

- Codice C:

```
A[12] = h + A[8]
```

  - `h` in `$s2`, indirizzo base di `A` in `$s3`
- Codice MIPS compilato:
  - L'indice 8 richiede un offset di 32 byte

```
lw  $t0, 32($s3)   # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)   # store word
```

Registri vs. Memoria

- I registri sono acceduti più velocemente della memoria
- Operare su dati in memoria richiede delle *load* e delle *store*
  - Più istruzioni da eseguire
- Il compilatore deve usare i registri per le variabili il più possibile
  - Rivolgersi alla memoria solo per variabili usate meno frequentemente
  - L'ottimizzazione dei registri è importante!

Operandi immediati

- Dati costanti sono specificati nell'istruzione stessa

```
addi $s3, $s3, 4
```

- Nessuna istruzione di sottrazione immediata
  - Basta usare una costante negativa

```
addi $s2, $s2, -1
```

- **Principio di Progettazione 3**: *"Rendere il caso comune veloce"*
  - Piccole costanti sono comuni
  - Gli operandi immediati evita un'istruzione di *load*

La costante Zero

- Il registro MIPS 0 (`$zero`) è la constante 0
  - Non può essere sovrascritto
- Utile per molte operazioni comune
  - Per esempio, spostamenti tra i registri

```
addi $t2, $t1, $zero
```

Interi binari senza segno

- Dato un numero su n bit

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Intervallo: da 0 a $2^n - 1$
- Esempio

  $\texttt{0000 0000 0000 0000 0000 0000 0000 1011}_2$

  $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Usando 32 bit
  - da 0 a $4'294'967'295$

Interi con segno in complemento a 2

- Dato un numero su n bit

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_1 2^1 + x_0 2^0$$

- Intervallo: da -2^{n-1} a $2^{n-1} - 1$
- Esempio

  $\texttt{1111 1111 1111 1111 1111 1111 1111 1100}_2$

  $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

  $= -2'147'483'648 + 2'147'483'644 = -4_{10}$

- Usando 32 bit
  - da $-2'147'483'648$ a $2'147'483'647$

Interi con segno in complemento a 2

- Il bit 31 è il bit del segno
  - 1 per numeri negativi
  - 0 per numeri non-negativi
- $-(-2^{n-1})$ non può essere rappresentato
- I numeri non-negativi hanno le stesse rappresentazioni in binario senza segno e in complemento a 2
- Alcuni numeri particolari:
  - 0: `0000 0000 ... 0000`
  - –1: `1111 1111 ... 1111`
  - Più negativo: `1000 0000 ... 0000`
  - Più positivo: `0111 1111 ... 1111`

Negazione con segno

- Complemento bit a bit a sommare 1
  - Complemento bit a bit significa $0 \rightarrow 1$, $1 \rightarrow 0$

$$x + \bar{x} = 111111....111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Esempio: negare $+2$
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

    $= 1111\ 1111\ \ldots\ 1110_2$

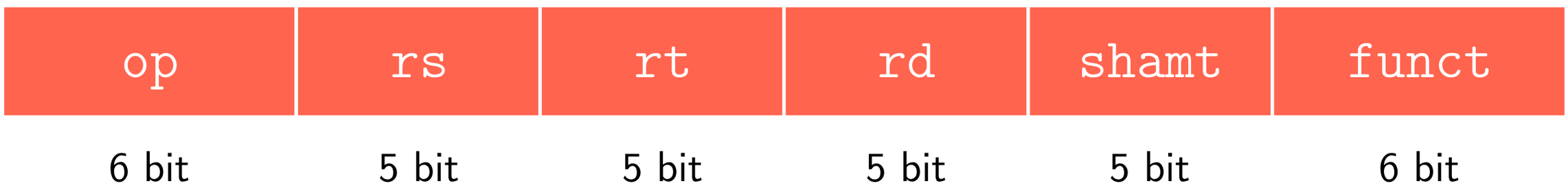
Estensione di segno

- Rappresentare un numero usando più bit
  - Mantenendo il valore numerico
  - Nell'instruction set del MIPS
    - `addi`: estendere il valore immediato
    - `lb, lh` : estendere il byte/halfword caricato
    - `beq, bne` : estendere lo spiazzamento
- Replicate il bit di segno a sinistra
  - per i valori senza segno: estendere con 0
- Esempi: da 8 bit a 16 bit
  - +2: 0000 0010 → 0000 0000 0000 0010
  - –2: 1111 1110 → 1111 1111 1111 1110

Rappresentare le istruzioni

- Le istruzioni sono codificate in binario
  - Sono chiamate codice macchina
- Istruzioni MIPS
  - Codificate come istruzioni word su 32 bit
  - Ridotto numero di formati per codificare il codice dell'operazione (opcode), numeri di registro, ...
  - Regolarità!
- Numeri di registro
  - `$t0` – `$t7` sono i registri 8 – 15
  - `$t8` – `$t9` sono i registri 24 – 25
  - `$s0` – `$s7` sono i registri 16 – 23

Istruzioni MIPS in formato R



- Campi dell'istruzione
- `op`: codice dell'operazione (opcode)
- `rs`: numero di registro della prima sorgente
- `rt`: numero di registro della seconda sorgente
- `rd`: numero di registro della destinazione
- `shamt`: shift amount (`00000` per ora)
- `funct`: codice della funzione (estende l'opcode)

Esempi formato R

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

`add $t0, $s1, $s2`

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010001100100100000000100000_2 = 02324020_{16}$

Esadecimale

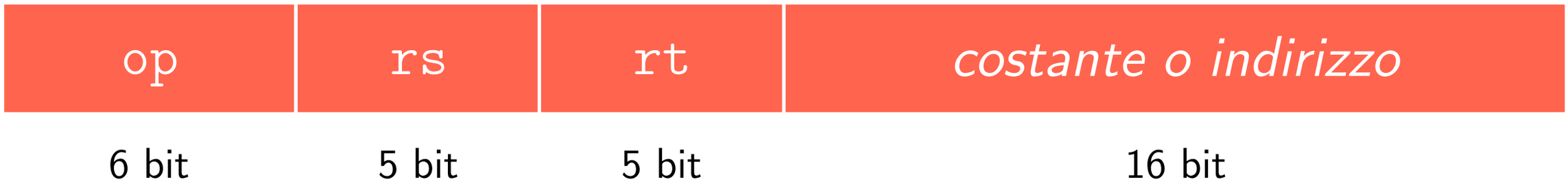
- Base 16
  - Rappresentazione compatta di stringhe di bit
  - 4 bit per cifra esadecimale

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Esempio: eca8 6420

```
1110 1100 1010 1000 0110 0100 0010 0000
```

Istruzioni MIPS in formato I



- Istruzioni load/store e aritmetiche immediate
  - `rt`: numero di registro della sorgente o destinazione
  - *costante*: da -2^{15} a $2^{15}-1$
  - *indirizzo*: offset aggiunto all'indirizzo base in `rs`
- **Principio di Progettazione 4**: *"un buon progetto richiede dei buoni compromessi"*
  - Formati differenti complicano la decodifica, ma permettono istruzioni su 32 bit uniformi
  - Mantenere i formati il più simile possibile